Jeannie User Guide

A compiler contributed to xtc, Version 1.13.3 (05/14/08)

Martin Hirzel and Robert Grimm

The current Jeannie project members are Robert Grimm, Martin Hirzel, Byeoncheol "BK" Lee, and Kathryn McKinley.

We received helpful feedback from Joshua Auerbach, Rodric Rabbah, Gang Tan, David Ungar, and Jan Vitek.

This material is based in part upon work supported by the National Science Foundation under Grants No. CNS-0448349 and CNS-0615129 and by the Defense Advanced Research Projects Agency under Contract No. NBCH30390004.

This is the user guide for a compiler contributed to xtc Version 1.13.3 (05/14/08).

Table of Contents

1 Introduction

Jeannie is a programming language that combines Java and C. It supports the full syntax of both languages, and adds a backtick (`) operator for nesting Java in C and nesting C in Java. You can use it to implement a feature of your Java application with an existing C library. In this case, you would write glue code that nests C in Java. Another common usage scenario is when you want to enhance your C application with some Java features, such as multi-threading, exception handling, or GUI controls. In this case, you would nest Java code in C.

The Jeannie language is implemented by a compiler contributed to xtc. That is the official name of the code that IBM has donated to the xtc compiler framework (http://cs.nyu.edu/rgrimm/xtc/). The compiler translates Jeannie code first into Java and C source code that uses the JNI, and then from there into class files for Java and a dynamically linked library for C. This user guide describes how to use the compiler and the language in practice. The research that went into Jeannie is described in a conference paper (http://domino.watson.ibm.com/comm/research_people.nsf/pages/hirzel.main.html#oopsla07-jeannie).

1.1 Installation

This section describes how to install xtc, which includes the Jeannie compiler, and how to test that the installed Jeannie compiler runs correctly.

1.1.1 Requirements

The Jeannie compiler uses Java Standard Edition version 5 or higher and several GNU command line tools, including gcc, bash, make, find, zip, and others. You need to make sure that the Java compiler, the JVM, and the GNU tools are installed and on your `PATH`. We have tested Jeannie with multiple Java virtual machines (IBM J9, Sun HotSpot, and Jikes RVM), and on multiple operating systems (Linux, Windows/Cygwin, and Mac OS X).

1.1.2 Download

You need xtc-core.zip to run Jeannie, xtc-testsuite.zip to test your local Jeannie installation, and antlr.jar and junit.jar to compile xtc. You can download these four files from their respective project websites, for example like this:

```
wget http://cs.nyu.edu/rgrimm/xtc/xtc-core.zip
wget http://cs.nyu.edu/rgrimm/xtc/xtc-testsuite.zip
wget http://www.antlr.org/download/antlrworks-1.1.4.jar
wget http://downloads.sourceforge.net/junit/junit-4.4.jar
```

Pick a directory where you want your local xtc and Jeannie installation to live. Assuming your directory is called `local_install_dir`, populate it with your downloads like this:

```
unzip -d local_install_dir xtc-core.zip
unzip -d local_install_dir xtc-testsuite.zip
mv antlrworks-1.1.4.jar local_install_dir/xtc/bin/antlr.jar
mv junit-4.4.jar local_install_dir/xtc/bin/junit.jar
```

1.1.3 Configuration

You need to set your `PATH` environment variable to include your Java 1.5 compiler and JVM. In addition, you need to set `PATH_SEP` either to '`:`' on Linux or Mac OS X or to '`;`' on Windows/Cygwin. Assuming you unzipped xtc to a directory called `local_install_dir`, you now need to perform the following steps:

```
export PATH_SEP=':'
export JAVA_DEV_ROOT=local_install_dir/xtc
export PATH=$JAVA_DEV_ROOT/src/xtc/lang/jeannie:$PATH
export CLASSPATH=$JAVA_DEV_ROOT/bin/junit.jar$PATH_SEP$CLASSPATH
export CLASSPATH=$JAVA_DEV_ROOT/bin/antlr.jar$PATH_SEP$CLASSPATH
export CLASSPATH=$JAVA_DEV_ROOT/classes$PATH_SEP$CLASSPATH
make -C $JAVA_DEV_ROOT classes configure
```

The last step will use `xtc/Makefile` to compile and configure xtc along with the Jeannie compiler. You may see some warning messages related to Java generics, but the compilation should keep going and finish without any fatal error messages.

1.1.4 Testing the installation

After completing the download and configuration step, try the following:

```
make -C $JAVA_DEV_ROOT check-jeannie
```

This first invokes a few hundred JUnit tests, each of which writes a dot '`.`' to the console. Next, it invokes a few dozen integration tests, each of which writes a couple of lines to the console. Overall, the testing output should look like this:

```
java -ea junit.textui.TestRunner xtc.lang.jeannie.UnitTests
.........................................
.........................................
```

many more dots for unit tests

```
.........................................
.............
Time: 7.15

OK (874 tests)

make -C local_install_dir/xtc/fonda/jeannie_testsuite cleanall
find local_install_dir/xtc/fonda/jeannie_testsuite -name '*~' -exec rm -f \{\} \;
rm -f -r tmp
rm -f core.*.dmp javacore.*.txt
make -C /Users/hirzel/local_install_dir/xtc/fonda/jeannie_testsuite test
==== integration test_000 ====
diff tmp/000mangled/output.txt tmp/000sugared/output.txt
==== integration test_001 ====
diff tmp/001mangled/output.txt tmp/001sugared/output.txt
```

many more lines for integration tests

```
==== integration test_035 ====
diff tmp/035mangled/output.txt tmp/035sugared/output.txt
==== integration test_036 ====
diff tmp/036mangled/output.txt tmp/036sugared/output.txt
==== integration tests completed ====
```

By the time you read this, there may be more tests than shown above. Two of the integration tests (18 and 26) write some timing numbers to the console, but as long as all tests end with `diff` and without finding any differences between the mangled and sugared output, everything went fine.

1.2 Hello world!

The following Jeannie program (integration test 041) has a Java main method that uses a nested C call to print `"Hello, world!"` to the console.

```
`.C {                                      // 1
#include <stdio.h>                         // 2
}                                          // 3
class Main {                               // 4
  public static void main(String[] args) { // 5
    `printf("Hello, world!\n");            // 6
  }                                        // 7
}                                          // 8
```

The file starts with a block of C code that includes a header file (Lines 1-3) containing, among other things, the prototype for the `printf` function. In general, the backtick symbol (`` ` ``) toggles between the languages Java and C. It can be either qualified (like `` `.C `` on Line 1) or simple (like on Line 6). The example code defines a Java class `Main` with a Java method `main` (Lines 4 and 5). The body of the method (Line 6) contains a simple backtick to toggle from Java to C for the call `printf("Hello, world!\n")`. When used in an expression, the backtick is a unary prefix operator that affects the following subexpression.

To test this hello world program, you need to compile it as follows:

```
jeannie.sh Main.jni
```

The Jeannie compiler will generate a class file (`Main.class`), a shared library (on Linux: `libMain.so`; on Windows/Cygwin: `Main.dll`; on Mac OS X: `libMain.jnilib`), and several intermediate files. Before you can run the example, you need to tell the operating system where to find the shared library, by adding the directory to your `PATH` and `LD_LIBRARY_PATH` environment variables. Assuming the example code is in the current directory (`.`), you can run it as follows:

```
java -cp . -Djava.library.path=. Main
```

This should, of course, print `"Hello, world!"` to the console.

1.3 Trouble shooting

As with any complex piece of software, you may run into trouble when trying to use the Jeannie compiler. This section describes a few common issues and how to address them. We will keep updating this section as we encounter additional difficulties and their solutions.

If you cannot compile the Jeannie compiler at all, or if it does not run, you should double-check whether all the required tools are installed on your local machine. In particular, you need Java 1.5 or higher, and you need the GNU C compiler, see Section 1.1.1 [Requirements], page 1. To get the required tools on Windows, use Cygwin. To get the required tools on Mac OS, install XCode (that should be on one of the CDs that came with your Mac), and get the remaining tools from an open source site such as Fink. Next, try whether you can run the tests that come with Jeannie, see Section 1.1.4 [Testing the installation], page 2.

Finally, double-check that you set your environment variables correctly, in particular, `PATH`, `CLASSPATH`, and `LD_LIBRARY_PATH`.

If the Jeannie compiler throws an internal exception rather than producing a nice error message, that's a bug; please report it, along with a minimal test case that reproduces it.

If you get your Jeannie program to compile, but it crashes at runtime, the two most common symptoms are segmentation faults or dynamic linker errors.

A segmentation fault occurs when your program tries to access an illegal memory address. This is usually caused by null pointers or out of bounds accesses in C. To find the defect, you should start by rebuilding your program from scratch, to rule out problems caused by inconsistent incremental compilation. Next, you should run with a symbolic interactive debugger; see Section 2.7 [Debugging], page 14. If you find that the problem is in the Jeannie compiler (e.g., using an illegal method ID), please report the bug, along with a minimal test case that reproduces it.

A dynamic linker error occurs when your program can not find a function that should be in a shared object file (DLL on Windows). This problem is common in hand-written JNI code, but should not occur for Jeannie-generated JNI code. To find the defect, you should start by rebuilding your program from scratch, to rule out problems caused by inconsistent incremental compilation. Next, make sure the shared library is on the path, using the `-Dload-library-path` JVM command line option or the `PATH` and `LD_LIBRARY_PATH` environment variables. If it still does not work, you should inspect the code related to the missing symbol. To make Jeannie-generated code easier to read, use the `-pretty` compiler flag. If you believe that the problem is caused by Jeannie-generated code, please report the bug, along with a minimal test case that reproduces it. It is more likely that the problem is caused by other shared libraries that you link to. Consult your local linker guru, and use tools such as `nm` to investigate the symbols of your object files. You may need to specify the external DLL like any other file at the end of the compiler command line.

2 Examples

This chapter discusses Jeannie by example. Each section uses a short self-contained piece of code to illustrate one aspect of how to use the Jeannie language. If you read this on your computer screen instead of in a printed hardcopy, I recommend you read the html version, since it does not have the page breaks of the pdf version. The examples are designed so you can easily copy-and-paste them and try them out yourself. You should play with the examples, changing things here and there to see what happens.

2.1 Program structure

The following example (integration test 039) illustrates the structure of a Jeannie file.

```
package cstdlib;                                                 // 1
import java.util.Random;                                         // 2
`.C {                                                            // 3
#include <math.h>                                                // 4
}                                                                // 5
class Math {                                                     // 6
  public static native double pow(double x, double y) `{         // 7
    return (`double)pow(`x, `y);                                 // 8
  }                                                              // 9
}                                                                //10
public class Main {                                              //11
  public static void main(String[] args) {                       //12
    Random random = new Random(123);                             //13
    for (int i=0; i<3; i++) {                                    //14
      double d = 100.0 * random.nextDouble();                    //15
      double r = Math.pow(d, 1.0 / 3.0);                         //16
      System.out.println("d " + d + " r " + r + " ^3 " + r*r*r); //17
    }                                                            //18
  }                                                              //19
}                                                                //20
```

A Jeannie file starts like a regular Java file with an optional package (Line 1) and imports (Line 2). These are followed by top-level C declarations enclosed in `` `.C{ } `` (Lines 3-5). They usually come from header files, as in the example, but you can also declare your own C functions and types in this section. The rest of the Jeannie file is structured like a regular Java file, with an optional package (Line 4), imports (Line 5), and one (Line 6) or more (Line 11) top-level classes or interfaces. The example illustrates how you might write a wrapper for parts of the C standard library, hence the package is called `cstdlib` (Line 4).

In Jeannie, a native method has a body, which must be a block of C code (Line 7). Inside of the C code, you can use backticked C types (such as `` `double `` on Line 8) that are equivalent to the corresponding Java types (e.g., `double`). You can also use nested Java expressions, for example, to refer to Java variables and parameters (such as `` `x `` and `` `y `` on Line 8).

To build this example, run the Jeannie compiler like this:

```
(bash) jeannie.sh -lm cstdlib/Main.jni
```

The -lm linker flag is passed on to the native C compiler, which uses it to link the m library (math) into the generated native shared object file. After compiling, the package directory cstdlib will contain class files for the top-level classes Math and Main, a shared library, and some compiler intermediate files. You can run the program like this:

```
(bash) java -cp . -Djava.library.path=./cstdlib cstdlib.Main
d 72.31742029971468 r 4.166272210190459 ^3 72.31742029971466
d 99.08988967772393 r 4.627464705142208 ^3 99.08988967772387
d 25.329310557439133 r 2.9368005732853377 ^3 25.32931055743913
```

The program should print a series of numbers with their cubic roots as shown above. You can simplify the command line by putting the shared library on your PATH or LD_LIBRARY_PATH.

The example illustrates how Jeannie toggles between the languages for a block, for an expression, or for a Java type name. In each case, you can use either the simple language toggle backtick (‘), or the qualified form (‘.C or ‘.Java). Language toggle is also allowed for certain Java statements in C (synchronized, try, throw), and for putting a throws clause on a C function. See Section 3.2.1 [Syntax], page 18, which shows the entire Jeannie grammar.

2.2 Locals

The following example (integration test 040) illustrates code with multiple local variables, both in Java (args, input, and hasDecimalPoint) and in C (intOrFloat, f, and i).

```
`.C { }                                                    // 1
class Main {                                               // 2
  public static void main(String[] args) {                 // 3
    String input = "12.34E1";                              // 4
    boolean hasDecimalPoint = -1 != input.indexOf('.'); // 5
    `.C {                                                  // 6
      `Number intOrFloat;                                  // 7
      if (`hasDecimalPoint) {                              // 8
        `Float f = `Float.valueOf(input);                  // 9
        intOrFloat = f;                                    //10
      } else {                                             //11
        `Integer i = `Integer.valueOf(input);              //12
        intOrFloat = i;                                    //13
      }                                                    //14
      `.Java {                                             //15
        System.out.println(`intOrFloat);                   //16
      }                                                    //17
    }                                                      //18
  }                                                        //19
}                                                          //20
```

You can run the program like this:

```
(bash) jeannie.sh Main.jni
(bash) java -cp . Main
123.4
```

Each local variable in Jeannie has a defining language (Java or C), a scope (a portion of the program text where it is valid), and a type. The following table characterizes the local variables from the example:

Java	`String[]`	`args`	3-18
Java	`String`	`input`	4-18
Java	`boolean`	`hasDecimalPoint`	5-18
C	`‘Number`	`intOrFloat`	7-17
C	`‘Float`	`f`	9-10
C	`‘Integer`	`i`	12-13

In Jeannie, you can only access a local variable in code of the same language. For example, Line 8 contains a C `if` statement, and must therefore toggle to Java to access the Java local variable `hasDecimalPoint`. And of course, you can only access a local variable if it is in scope; for example, the scope of `intOrFloat` ends in Line 17, so the variable can not be used after that. Like in Java and C, scopes can nest, and variables in inner scopes can shadow variables of the same name from outer scopes. Backticked expressions in Jeannie are immutable (in programming languages terminology, they are not l-values, since they can not appear on the left-hand side of an assignment). That means that any modification to a variable has to occur in the variable's language.

This example also illustrates that C local variables can hold references to Java objects. For example, the result of `‘Float.valueOf(..)` in Line 9 is a reference to a Java object containing a boxed floating point number. This reference gets stored in the C local variable `f`. Note that this variable has type `‘Float`. In Jeannie, a backticked Java type is a C type. Furthermore, since class `Float` is a subclass of `Number` in Java, Jeannie permits the C code in Line 10 to widen the reference in the assignment `intOrFloat = i`. On the other hand, if the code were to contain the reverse assignment `i = intOrFloat`, the compiler would give an error message. You should try it out.

2.3 Garbage collection

Do not store references to Java objects in non-local C data. Non-local data is any data that is not in local variables, and thus, does not go away when the enclosing function or method returns. In other words, non-local data in C resides in global variables or on the heap. You should not store any references to Java objects there, because by the time you access them again, the objects may have already been garbage collected. When that happens, the reference is a dangling reference, and using it can cause a crash, or worse, can corrupt important data. In fact, on some JVMs, the reference is unusable even without garbage collection, which make the problem easier to diagnose, because the program fails more quickly and deterministically.

Instead, you should store references to Java objects into Java static or instance fields. Jeannie makes it very easy to access a Java field from C by using a backtick. The following example illustrates the difference between storing reference to a Java object in a C global variable versus a Java static field.

```
import java.io.PrintWriter;                                         // 1
‘.C {                                                               // 2
‘PrintWriter badGlob;                                               // 3
}                                                                   // 4
```

```
class Main {                                                          // 5
  static PrintWriter goodGlob;                                        // 6
  static native void setGlob(boolean beGood, PrintWriter init) `{ // 7
    if (`beGood) `( Main.goodGlob = init );                          // 8
    else          badGlob = `init;                                     // 9
  }                                                                   //10
  static native PrintWriter getGlob(boolean beGood) `{               //11
    if (`beGood) return `Main.goodGlob;                               //12
    else          return badGlob;                                     //13
  }                                                                   //14
  static native void useGlob(boolean beGood, Object obj) `{          //15
    `.Java {                                                          //16
      PrintWriter out = Main.getGlob(beGood);                         //17
      out.println(obj);                                               //18
      out.flush();                                                    //19
    }                                                                 //20
  }                                                                   //21
  public static void main(String[] args) {                            //22
    boolean beGood = true;                                            //23
    setGlob(beGood, new PrintWriter(System.out));                     //24
    for (int i=0; i<3; i++) {                                         //25
      useGlob(beGood, "o_" + i);                                      //26
      System.gc();                                                    //27
    }                                                                 //28
  }                                                                   //29
}                                                                     //30
```

If you run this program unchanged, it uses a Java static field, if you change Line 23 to set `beGood = false`, it uses a C global variable. In the good case, it prints `o_0 o_1 o_2`, otherwise, it crashes with an error message that depends on your Java virtual machine, operating system, and C compiler. You should try it out, so you can recognize the error if you see the symptom again in another context. You should also try whether the symptom goes away if you delete Line 27.

If you do make a mistake related to global references, you may end up needing a debugger to find the source of the defect; see Section 2.7 [Debugging], page 14.

2.4 Arrays

This section is about how C code can access Java arrays. Arrays are important for Jeannie, since people frequently use native code either for I/O, which usually involves buffers, or for high-performance computing, which usually involves matrix computations. Just like any other Java expression can be nested in C using a backtick, so can Java expressions that access an array. C code can read from a Java array using a Java array subscript, for instance, `` `arr[i] ``. C code can write to a Java array using a Java assignment, for instance, `` `(arr[i] = v) ``. Since backticked expressions in Jeannie are immutable, a C assignment to a Java array (e.g., `` `arr[i] = v ``) would be illegal.

The following example (integration test 043) shows a native method `replace(chars, oldC, newC)` that modifies the Java array `chars`, replacing the first occurrence of `oldC` in chars by newC. It returns the index of the replaced element, or -1 if the element was not found.

```
‘.C{ }
class Main {
  static native int replace(char[] chars, char oldC, char newC) ‘{
    for (int i=0; i<‘chars.length; i++) {
      if (‘oldC == ‘chars[‘i]) {
        ‘(chars[‘i] = newC);
        return (‘int)i;
      }
    }
    return (‘int)-1;
  }
  public static void main(String []args) {
    char[] a = { ’a’, ’b’, ’c’ };
    int r;
    r = replace(a, ’b’, ’d’);
    System.out.println(r + "  " + new String(a));
    r = replace(a, ’b’, ’d’);
    System.out.println(r + "  " + new String(a));
  }
}
```

The example also includes a main method that invokes replace twice to replace ’b’ by ’d’. You can compile and run the program like this:

```
(bash) jeannie.sh Main.jni
(bash) java -cp . Main
1  adc
-1  adc
```

The output shows that the first call to replace changed the element at index 1, yielding adc, whereas the second call did not find any element to change and therefore returned -1, leaving the array unchanged as adc.

Accessing arrays with simple backticked Java expressions is convenient. But users may want to use Java arrays in performance-critical loops, where the transition between languages can become a bottle-neck. To accommodate faster access to an entire Java array, Jeannie provides the with-statement. The header of a with-statement associates a C variable with a Java array; for example, with(‘char* s = ‘chars) { .. } associates the C variable s with the Java array chars. The body of the with statement can use that C variable as a normal C array. For example, the following code (integration test 044) implements the same replace method as before, but this time using a with-statement instead of a simple array access. Notice that the body of the for-loop is pure C code without language transitions.

```
‘.C{ }
class Main {
  static native int replace(char[] chars, char oldC, char newC) ‘{
    ‘char old = ‘oldC, new = ‘newC;
```

```
    ‘int len = ‘chars.length;                                        // 5
    with (‘char* s = ‘chars) {                                       // 6
      for (int i=0; i<len; i++) {                                    // 7
        if (old == s[i]) {                                           // 8
          s[i] = new;                                                // 9
          return (‘int)i;                                            //10
        }                                                            //11
      }                                                              //12
      cancel s;                                                      //13
    }                                                                //14
    return (‘int)-1;                                                 //15
  }                                                                  //16
  public static void main(String []args) {                           //17
    char[] a = { ’a’, ’b’, ’c’ };                                    //18
    int r;                                                           //19
    r = replace(a, ’b’, ’d’);                                        //20
    System.out.println(r + "  " + new String(a));                    //21
    r = replace(a, ’b’, ’d’);                                        //22
    System.out.println(r + "  " + new String(a));                    //23
  }                                                                  //24
}                                                                    //25
```

The `main`-method is unchanged, and this program should produce the same output as the previous example. In general, the initializer of a `with`-statement can be a variable declaration, like in the example, or an assignment. The types of the C variable and the Java expression must match: if the C variable has type ‘*E*`*`, the Java expression must have type `j`*E*`Array`.

Changes to the C array are reflected back to the Java array when control leaves the `with` statement, unless the user decided to `cancel` the changes, or there was an exception. In those cases, the Java array remains unchanged. In the example, Line 10 leaves the `with`-statement and the method, at which time Jeannie makes any pending modifications to the Java array `chars`. Line 13 also leaves the `with`-statement, but Jeannie drops any changes that may have occurred in the array.

So far, this section has focused on cases where C code wants to work directly with Java arrays. Jeannie supports that by simple nested Java expressions, and by the `with` statement for bulk accesses. But there are other cases where C code wants to copy just (parts of) an array between Java and C. Jeannie supports that with a pair of builtin functions `copyFromJava` and `copyToJava`. They have the following signatures:

```
‘int copyFromJava(‘E* ca, ‘int ci, jEArray ja, ‘int ji, ‘int len)
‘int copyToJava(jEArray ja, ‘int ji, ‘E* ca, ‘int ci, ‘int len)
```

In both cases, the return value is the number of copied elements. In both cases, the parameter list starts with the destination array and start index, followed by the source array and start index, followed by the number of elements to be copied. The following example (integration test 045) reimplements our familiar `replace` method using the trans-lingual copy functions.

```
‘.C{ }                                                               // 1
```

```
class Main {                                                           // 2
  static native int replace(char[] chars, char oldC, char newC) `{ // 3
    `char old = `oldC, new = `newC;                                    // 4
    `int len = `chars.length;                                          // 5
    `char s[len];                                                      // 6
    copyFromJava(s, 0, `chars, 0, len);                                // 7
    for (int i=0; i<len; i++) {                                        // 8
      if (old == s[i]) {                                               // 9
        s[i] = new;                                                    //10
        copyToJava(`chars, 0, s, 0, len);                              //11
        return (`int)i;                                                //12
      }                                                                //13
    }                                                                  //14
    return (`int)-1;                                                   //15
  }                                                                    //16
  public static void main(String []args) {                             //17
    char[] a = { 'a', 'b', 'c' };                                      //18
    int r;                                                             //19
    r = replace(a, 'b', 'd');                                          //20
    System.out.println(r + "  " + new String(a));                      //21
    r = replace(a, 'b', 'd');                                          //22
    System.out.println(r + "  " + new String(a));                      //23
  }                                                                    //24
}                                                                      //25
```

Again, the `main` method is unchanged, and the console output is the same as in the previous two examples. See Section 3.3.1 [copyFromJava], page 25 and Section 3.3.2 [copyToJava], page 25 for reference documentation on the two functions.

2.5 Abrupt control flow

Control flow is the order in which code executes. Normal control flow occurs when statements execute in the order in which they appear in the program, as well as when code has conditionals, loops, and calls. Abrupt control flow occurs when control jumps suddenly, for example because of a `return` statement in the middle of a function or method. Jeannie supports all the abrupt control flow constructs of Java and C (`return`, `break`, `continue`, `goto`, implicit exceptions, and explicit `throw`) and two new abrupt control flow constructs for bulk array manipulation (`commit`, `cancel`).

You can use Jeannie to obtain Java exception handling for C code. To throw a Java exception from C, use a nested Java `throw` statement. To handle a Java exception from C, use nested C handlers in a Java `try`/`catch`/`finally` statement. Jeannie implements the expected abrupt control flow. It also takes care of releasing internal resources. For example, a Jeannie `with` statement can allocate a temporal C array to cache Java data; if there is an exception during the `with` statement, Jeannie releases the temporary array.

The following example (integration test 046) illustrates abrupt control flow in Jeannie.

```
`.C {                                                // 1
#include <stdio.h>                                   // 2
```

```
}                                                      // 3
class Main {                                           // 4
  public static void main(String[] args) {             // 5
    int[] ja = { 1, 2, 3, 0 };                         // 6
    `.C {                                              // 7
      FILE* out;                                       // 8
      `try `{                                          // 9
        out = fopen("out.txt", "w");                   //10
        with (`int* ca = `ja) {                        //11
          for (`int i=0; i<4; i++) {                   //12
            if (ca[i] == 0)                            //13
              `throw new ArithmeticException("/ by 0"); //14
            ca[i] = 10 / ca[i];                        //15
            fprintf(out, "ca[%ld] == %ld\n", i, ca[i]); //16
          }                                            //17
        }                                              //18
      } catch (ArithmeticException e) `{               //19
        fprintf(out, "division by zero\n");            //20
      } finally `{                                     //21
        fclose(out);                                   //22
      }                                                //23
    }                                                  //24
    for (int i=0; i<4; i++)                            //25
      System.out.println("ja[" + i + "] == " + ja[i]);  //26
  }                                                    //27
}                                                      //28
```

The C code divides 10 by every number in an array, and writes the results to a file `out.txt`. At the end, the Java code writes the array contents to the console. When you compile and run this program, you should see the following:

```
(bash) jeannie.sh Main.jni
(bash) java -cp . Main
ja[0] == 1
ja[1] == 2
ja[2] == 3
ja[3] == 0
(bash) cat out.txt
ca[0] == 10
ca[1] == 5
ca[2] == 3
division by zero
```

The C code in Lines 7 thru 24 operates on a file `out`. Line 10 opens the file for writing, and Line 22 closes it again. To guarantee that the file gets closed no matter what happens, Line 10 is in a `try`-block and Line 22 is in the associated `finally`-block.

The C code in Lines 11 thru 18 operates on a C version `ca` of the Java array `ja`. Line 15 modifies the C array, and Line 16 prints the modification to the file. The original array from Line 6 is `{1,2,3,0}`, and Line 15 modifies it to `{10/1,10/2,10/3,..}`, yielding the result {10,5,3,..}. However, when the loop reaches the array element 0, Line 14 throws an exception to prevent division by zero. In Jeannie, an exception in a `with` statement cancels the modifications to the Java array. Therefore, when Lines 25 and 26 print `ja`, they observe the original contents from Line 6, namely {1,2,3,0}.

Jeannie does not permit `break`, `continue`, or `goto` to cross the language boundary or to leave a `with` statement, since that would yield to ill-defined behavior.

2.6 Strings

Jeannie supports access from C code to Java strings similarly to its support for arrays, with three important differences:

- C code can not copy elements to Java strings, since they are immutable.
- C code can access either Java's UTF-16 encoding of strings, or a UTF-8 encoding.
- The builtin functions `newJavaString` and `stringUTFLength` facilitate common string processing tasks.

The following example (integration test 047) demonstrates Jeannie's string manipulation features. Class `cstdlib.StdIO` is a simple wrapper for the functions `fputs` and `fflush` from the C `stdio` library, and class `cstdlib.TestDriver` exercises the code.

```
package cstdlib;                                                // 1
import java.io.IOException;                                     // 2
`.C {                                                           // 3
#include <stdio.h>                                              // 4
#include <errno.h>                                              // 5
#include <string.h>                                             // 6
}                                                               // 7
class StdIO {                                                   // 8
  public static native int stdOut() `{                          // 9
    return (`int)stdout;                                        //10
  }                                                             //11
  public static native void                                     //12
  fputs(String s, int stream) throws IOException `{             //13
    `int len = stringUTFLength(`s);                             //14
    `byte cs[1 + len];                                          //15
    int result;                                                 //16
    copyFromJava(cs, 0, `s, 0, `s.length());                    //17
    cs[len] = '\0';                                             //18
    result = fputs((char*)cs, (FILE*)`stream);                  //19
    if (EOF == result)                                          //20
      `throw new IOException(`newJavaString(strerror(errno))); //21
  }                                                             //22
  public static native void                                     //23
  fflush(int stream) throws IOException `{                      //24
    int result = fflush((FILE*)`stream);                        //25
    if (EOF == result)                                          //26
      `throw new IOException(`newJavaString(strerror(errno))); //27
  }                                                             //28
```

```
}                                                               //29
public class Main {                                             //30
  public static void main(String[] args) throws IOException {  //31
    StdIO.fputs("Schöne Grüße!\n", StdIO.stdOut());            //32
    StdIO.fflush(StdIO.stdOut());                              //33
  }                                                             //34
}                                                               //35
```

You can compile and run this program as follows:

```
(bash) jeannie.sh cstdlib/Main.jni
(bash) java -cp . -Djava.library.path=cstdlib cstdlib.Main
Sch\313\206ne Gr\302\270\357\254\202e!
```

Line 17 uses the builtin function `copyFromJava` to copy the Java string `s` to the C array `cs`. Here, this function behaves slightly differently from when we saw it in Section 2.4 [Arrays], page 8. Since the target of the copy is an array not of `‘char` but of `‘byte`, Line 17 performs a conversion from UTF-16 to UTF-8 encoding for unicode. In this example, the input string is `"Schöne Grüße!\n"` ("Nice greetings!" in German), which has 14 characters, including the Umlauts ö, ü, and ß. These special symbols take only 1 UTF-16 character each, but multiple UTF-8 bytes, hence the length of the resulting string `"Sch\313\206ne Gr\302\270\357\254\202e!"` is 18. Jeannie provides a function `stringUTFLength` that you can use to find out the number of bytes that a UTF-8 string will need before you make the conversion from UTF-16. In the example, Line 14 calls `stringUTFLength`, and Line 15 uses the result to stack-allocate a buffer for the C string. Note that the buffer has one more byte, used to zero-terminate the string in Line 18 as expected by the C language.

Lines 20 and 21 perform error handling. If the call to the C function `fputs` in Line 19 failed, it returns `EOF` to indicate that something went wrong. In that case, `errno` contains a numerical error code, and `strerror(errno)` describes the error as a C string. Line 21 converts that C string to a Java string with the Jeannie builtin function `newJavaString`, and then throws an `IOException`.

Besides the functions `copyFromJava`, `stringUTFLength`, and `newJavaString` illustrated in this example, Jeannie also supports strings in `with` statements. Since Java strings are immutable, you can not modify a Java string with a `with` statement either: it always implicitly cancels.

2.7 Debugging

We are actively working on a Jeannie debugger. In the meantime, we recommend you use gdb, following these instructions by Matthew White: `http://www.ibm.com/developerworks/java/library/j-jnidebug/index.html`.
Here is a short summary of Matthew White's approach. Essentially, you need to run the compiler with `-g` and the Java virtual machine with `-Xrunjdwp`. Then, you need to attach `jdb` and `gdb` to the running Java virtual machine. Then, at any given point, the system is in one of three states:

JVM active, and both `jdb` and `gdb` inert

: The Java virtual machine is active executing Java code, and both debuggers (`jdb` and `gdb`) are inert. This continues until either one of the debuggers hits a breakpoint, or there is a segmentation fault that activates `gdb`, or the program terminates.

`gdb` active, and both JVM and `jdb` inert
: The C debugger is active, allowing you to interact with it using debugging commands such as single-stepping, inspecting the C stack backtrace, inspecting C variable values, or setting C breakpoints. The JVM is suspended, and the Java debugger is inert. To get back into the first state (JVM active), ask the debugger to let the program continue.

`jdb` active, and both JVM and `gdb` inert
: The Java debugger is active, allowing you to interact with it using debugging commands such as single-stepping, inspecting the Java stack backtrace, inspecting Java variable values, or setting Java breakpoints. The JVM is suspended, and the C debugger is inert. To get back into the first state (JVM active), ask the debugger to let the program continue.

Consider the following buggy Jeannie program (integration test 048):

```
`.C {                                       // 1
int decr(int x) {                           // 2
  int y;                                    // 3
  x--;                                      // 4
  if (x != 0)                               // 5
    y = x;                                  // 6
  return y;                                 // 7
}                                           // 8
}                                           // 9
class Main {                                //10
  public static void main(String[] args) { //11
    int z = 1;                              //12
    z = `decr(`z);                          //13
    System.err.println(z);                  //14
  }                                         //15
}                                           //16
```

Since function `f` is called with `x==1`, the variable `y` is not initialized when Line 7 returns it. Thus, the uninitialized value taints variable `z` on Line 13, and Line 14 prints it. Below is an example debugging session, following Matthew White's approach. The session actually takes place in three different terminals, we interleave it here in chronological order for clarity. Lines marked with `*` contain user input.

```
-------- JVM terminal --------
* (bash) jeannie.sh -g Main.jni
* (bash) java -cp . -Xdebug -Xnoagent -Djava.compiler=none \
*   -Xrunjdwp:transport=dt_socket,server=y,suspend=y Main
  Listening for transport dt_socket at address: 50067
-------- jdb terminal --------
* (bash) jdb -attach 50067
  Set uncaught java.lang.Throwable
  Set deferred uncaught java.lang.Throwable
```

```
  Initializing jdb ...
  VM Started: No frames on the current call stack
* main[1] stop in Main.main
  Deferring breakpoint Main.main.
  It will be set after the class is loaded.
* main[1] run
  > Set deferred breakpoint Main.main
  Breakpoint hit: "thread=main", Main.main(), line=11 bci=0
  11      public static void main(String[] args) { //11
-------- gdb terminal --------
* (bash) ps -A | grep java | grep -v grep
  5980  p1  S+  0:00.16 java -cp . -Xdebug -Xnoagent -Djava.compiler=...
* (bash) gdb -quiet java 5980
  Attaching to program: `/usr/bin/java', process 5980.
  Reading symbols for shared libraries ........................... done
  0x90009cd7 in mach_msg_trap ()
* (gdb) break Main.jni:4
  Breakpoint 1 at 0x25e0d5f: file /Users/hirzel/tmp/Main.jni, line 4.
* (gdb) cont
  Continuing.
  [Switching to process 5980 thread 0xc07]
-------- jdb terminal --------
* main[1] cont
-------- gdb terminal --------
  Breakpoint 1, decr (x=1) at /Users/hirzel/tmp/Main.jni:4
  4   x--;                                      // 4
* (gdb) print y
  $1 = 39718276
* (gdb) cont
  Continuing.
-------- JVM terminal --------
  39718276
-------- jdb terminal --------
  The application exited
-------- gdb terminal --------
  Program exited normally.
```

3 Reference

Use this section to look up descriptions of Jeannie features, builtin functions, and tools.

3.1 Quick reference

Features

File = [*Java.Package*] *Java.Imports* `` `.C { `` *C.Declarations* `}` *Java.TypeDecls*
: A Jeannie file is a Java file starting with a block of C declarations.

Java.NT `+=` . . . / (`` ` `` / `` `.C ``) *C.NT*
: C blocks or expressions can be nested in Java code using backticks.

C.NT `+=` . . . / (`` ` `` / `` `.Java ``) *Java.NT*
: Java blocks or expressions can be nested in C code using backticks.

C.TypeSpecifier `+=` . . . / (`` ` `` / `` `.Java ``) *Java.Type*
: A backticked Java type name can serve as a C type specifier.

C.Statement `+=` . . . / `_with` `(` (*C.Assignment* / *C.Declaration*) `)` *C.Block*
: A `with` statement provides bulk access to a Java string or array.

C.Statement `+=` . . . / `_cancel` *C.Identifier* `;` / `_commit` *C.Identifier* `;`
: Cancel or commit end the `with` statement for the C variable name.

Builtin functions

`` `int _copyFromJava( ``*CT* `` ca, `int ci, ``*JT* `` ja, `int ji, `int len) ``
: Copy string or array elements from Java to C.

`` `int _copyToJava( ``*JT* `` ja, `int ji, ``*CT* `` ca, `int ci, `int len) ``
: Copy array elements from C to Java.

`` `String _newJavaString(const ``*CT* `ca)`
: Create a new Java string from C.

`` `int _stringUTFLength(`String js ``[`` , `int ji, `int len ``] `)`
: Count length of Java string in UTF-8 representation.

Tools

`jeannie.sh` [*options*] *file* [*c-files...*]
: Master script. Creates dynamically linked library and class files.

`java xtc.lang.jeannie.Preprocessor` [*options*] *file*
: Inject Jeannie-specific definitions. Output goes to `stdout`.

`java xtc.lang.jeannie.Jeannie` [`-analyze | -translate |` . . .] *file*
: Translate preprocessed Jeannie source code to Java and C source.

`java xtc.lang.ClassfileSourceRemapper` [*options*] *source-file class-file*
: Add debugging symbols to classes. Rewrites the *class-file*.

3.2 Language features

This section discusses the syntax and semantics of Jeannie, first in summary and then individually by feature.

3.2.1 Syntax

Below is the Jeannie grammar. It has four groups of productions: the start symbol, modifications to the Java and C grammars, and additions to the C grammar. Each grammar production consists of a non-terminal, followed by "=", "+=", or ":=", followed by a parsing expression. For example, the production for the start symbol

File = [*Java.Package*] *Java.Imports* ‘.C { *C.Declarations* } *Java.TypeDecls*

specifies that the non-terminal *File* recognizes an optional package declaration, import declarations, some initial *C.Declarations* enclosed in a ‘.C{...} block, and finally top-level Java class and interface declarations. Each grammar production is followed by an example expansion. In the case of *File*, the example expansion is

↦ ‘.C { #include <stdio.h> } class A { }

Productions with "+=" modify the grammar of one of the two base languages with the grammar modification facilities of Rats!. For example,

Java.Block += . . . / *CInJava C.Block*

modifies the Java grammar: the non-terminal *Java.Block*, in addition (+=) to recognizing Java blocks (. . .), now recognizes a backtick (*CInJava*) followed by a C block (*C.Block*). As another example the rule

C.FunctionDeclarator := *C.DirectDeclarator* (*C.ParameterDeclaration*)
[*JavaInC Java.ThrowsClause*]

modifies the C grammar: the non-terminal *C.FunctionDeclarator*, instead of (:=) recognizing just a C function declarator, now recognizes a C function declarator followed by an optional backtick and Java `throws` clause.

Start symbol

File = [*Java.Package*] *Java.Imports* ‘.C { *C.Declarations* } *Java.TypeDecls*
↦ ‘.C { #include <stdio.h> } class A { }

Modifications to Java grammar

Java.Block += . . . / *CInJava C.Block*
↦ ‘{ int x = 42; printf("%d", x); }

Java.UnaryExpression += . . . / *CInJava C.UnaryExpression*
↦ ‘((jboolean)feof(stdin))

CInJava = ‘.C / ‘
↦ ‘.C

Modifications to C grammar

C.Block += . . . / *JavaInC Java.Block*
↦ ‘{ int x=42; System.out.println(x); }

C.UnaryExpression += . . . / *JavaInC Java.UnaryExpression*
↦ ‘new HashMap();

C.TypeSpecifier += . . . / *JavaInC Java.Type*
    ↦ `‘java.util.Map`

C.FunctionDeclarator `:=` *C.DirectDeclarator* `(` *C.ParameterDeclaration* `)`
    [*JavaInC Java.ThrowsClause*]
        ↦ `f(char *s) ‘throws IOException`

C.Statement += . . .
    / *JavaInC Java.SynchronizedStatement*
        ↦ `‘synchronized(act) { act.deposit(); }`
    / *JavaInC Java.TryStatement*
        ↦ `‘try { f(); } catch (Exception e) { h(e); }`
    / *JavaInC Java.ThrowStatement*
        ↦ `‘throw new Exception("boo");`

JavaInC = `‘.Java` / `‘`
    ↦ `‘.Java`

Additions to C grammar

C.Statement += . . .
    / `_with` `(` *WithInitializer* `)` *C.Block*
        ↦ `_with (‘int* ca = ‘ja) { sendMsg(ca); }`
    / `_cancel` *C.Identifier* `;`
        ↦ `_cancel ca;`
    / `_commit` *C.Identifier* `;`
        ↦ `_commit ca;`

WithInitializer =
    C.AssignmentExpression
        ↦ `msg->data = ‘ja`
    / *C.Declaration*
        ↦ `‘int* ca = ‘v.toArray()`

3.2.2 Type equivalences

Jeannie introduces new C types for every Java primitive, class, or interface type. If *JT* is a Java type name, then ‘*JT* is a C type. For example, `‘int` is a signed 32-bit C integer type, and `‘java.io.IOException` is the type for opaque C references to Java IOException objects.

Jeannie defines several type equivalences between Java and C types, denoted as $JT \equiv CT$. When a Java expression is nested in C code, Jeannie type-checks the C code as if the Java expression had the equivalent C type. Likewise, when a C expression is nested in Java code, Jeannie type-checks the Java code as if the C expression had the equivalent Java type. Of course, each Java primitive, class, or interface type is equivalent to the same type with backtick in C. For example:

`int`	≡	`‘int`
`java.io.IOException`	≡	`‘java.io.IOException`

```
java.util.Iterator     ≡ `java.util.Iterator
```

Jeannie has the same rules for resolving simple type names to fully qualified names as Java. For example, C code in Jeannie can use `` `IOException `` for `` `java.io.IOException `` if the current file is part of package `java.io` or if it has the appropriate import declaration.

In addition to backticked Java types in C, Jeannie also honors type equivalences between Java and C types from `jni.h`. The most important ones are arrays:

```
boolean[]              ≡ jbooleanArray
byte[]                 ≡ jbyteArray
char[]                 ≡ jcharArray
short[]                ≡ jshortArray
int[]                  ≡ jintArray
long[]                 ≡ jlongArray
float[]                ≡ jfloatArray
double[]               ≡ jdoubleArray
java.lang.Object[]     ≡ jobjectArray
```

Other type equivalences from `jni.h` include primitive types and certain frequently used classes and interfaces:

```
boolean                ≡ jboolean
byte                   ≡ jbyte
char                   ≡ jchar
short                  ≡ jshort
int                    ≡ jint
long                   ≡ jlong
float                  ≡ jfloat
double                 ≡ jdouble
java.lang.Object       ≡ jobject
java.lang.Class        ≡ jclass
java.lang.String       ≡ jstring
java.lang.Throwable    ≡ jthrowable
```

C pointers, structs, and unions have no equivalent in Java, and the Jeannie compiler flags an error when a program attempts to use them in Java code.

3.2.3 C in Java

NAME C in Java – C block or C expression nested in Java code.

SYNTAX RULES

Java.Block	+=	. . . / *CInJava C.Block*
Java.UnaryExpression	+=	. . . / *CInJava C.UnaryExpression*
CInJava	=	`` `.C `` / `` ` ``

SYNTAX NOTES

An example for a C block in Java is `` `{ int x = 42; printf("%d", x); } ``.

An example for a C expression in Java is `` `((jboolean)feof(stdin)) ``.

When used in an expression, the backtick (`` ` `` or `` `.C ``) has the same precedence as other unary prefix operators such as logical negation (`!`).

Dynamic semantics

The dynamic semantics of nested C code are mostly just the dynamic semantics of C. All C code in the same activation of a function or method observes the same state, so when nested blocks or expressions have side effects, those effects are visible to other code, as expected. It is a programmer mistake to keep a reference to a Java object into a C global variable or the C heap after the current function or method returns. If a Java exception occurs in the nested C code, control abruptly leaves the nested C code and propagates to the nearest Java exception handler, following the dynamic semantics of Java. Java code that contains a nested C expression uses the result of that C expression as an r-value of the corresponding Java type.

Static semantics

The static semantics of nested C code are mostly just the static semantics of C. For instance, Jeannie resolves C functions to their prototypes, which are typically declared in header files included at the beginning of a Jeannie file. If name lookup fails or the C code is incorrectly typed, the Jeannie compiler reports an error.

When a C expression is nested in Java code, Jeannie type-checks the Java code as if the C expression had the equivalent Java type, as specified in Section 3.2.2 [Type equivalences], page 19. It is a compile time error when a C expression nested in Java evaluates to a pointer, struct, or union, since those have no equivalent in Java. Jeannie also checks that the Java code treats the value of the C expression as an r-value, and in particular, does not assign to it. When a C `return` statement returns from a Java method, Jeannie type-checks the return value against the return type of the method as if it had the equivalent Java type.

C assignments, variable initializers, function invocations, and `return` statements can implicitly widen opaque references to Java classes or interfaces. For example, C code can assign a reference of type `` `java.util.HashMap `` to a variable of type `` `java.util.Map ``, because class `HashMap` implements interface `Map`.

Native methods of a Java class must have a body and that body must be a backticked C block. Native methods also declare an implicit C parameter `JNIEnv* env`, so that C code has access to JNI's API. Consequently, explicit parameters of native methods cannot have the name `env`. Jeannie provides this feature to facilitate incremental conversion of JNI code to Jeannie; other uses of this feature are discouraged.

If nested C code contains any `break`, `continue`, or `goto` statements, those must not cross the language boundary, and they also must not cross the boundary of a `with` statement.

3.2.4 Java in C

Name Java in C – Java block, expression, or other code nested in C code.

Syntax rules

C.Block	+=	. . . / *JavaInC Java.Block*
C.UnaryExpression	+=	. . . / *JavaInC Java.UnaryExpression*

C.TypeSpecifier	`+=`	. . . / *JavaInC Java.Type*
C.Statement	`+=`	. . .
	/	*JavaInC Java.SynchronizedStatement*
	/	*JavaInC Java.TryStatement*
	/	*JavaInC Java.ThrowStatement*
C.FunctionDeclarator	`:=`	*C.DirectDeclarator*
		`(` *C.ParameterDeclaration* `)`
		[*JavaInC Java.ThrowsClause*]
JavaInC	=	`` `.Java `` / `` ` ``

Syntax notes

An example for a Java block in C is `` `{ int x=42; System.out.println(x); } ``.

An example for a Java expression in C is `` `new HashMap(); ``.

An example for a Java type name in C is `` `java.util.Map ``. It may be part of a C variable declaration such as `` const `java.util.Map m = ...; ``.

An example for a Java statement in C is `` `throw new Exception("boo"); ``.

An example for a C function declarator with a Java throws clause is `` f(char *s) `throws IOException ``.

When used in an expression, the backtick (`` ` `` or `` `.C ``) has the same precedence as other unary prefix operators such as logical negation (`!`).

Dynamic semantics

The dynamic semantics of nested Java code are mostly just the dynamic semantics of Java. The semantics of exceptions and locks extend from Java across C code; for example, when an exception abruptly leaves a synchronized statement, the corresponding lock is released. All Java code in the same activation of a function or method observes the same state, so when nested Java code has side effects, those effects are visible to other code, as expected.

C code that contains a nested Java expression uses the result of that Java expression as an r-value of the corresponding C type. As specified in Section 3.2.2 [Type equivalences], page 19, the corresponding C type may be a backticked Java primitive, class, or interface type. If a nested Java expression yields a reference to a Java object, that object will not be garbage collected until at least the enclosing function or method returns. In the terminology of JNI, it constitutes a local reference.

Static semantics

The static semantics of nested Java code are mostly just the static semantics of Java. For instance, Jeannie resolves Java class names relative to imports. It also verifies that all checked exceptions are either caught locally or declared as thrown by the enclosing function or method. Furthermore, Jeannie checks that Java members are in fact accessible, i.e., that references to fields, methods, and member types obey their visibility (`private`, `protected`, `public`, or default).

When a Java expression is nested in C code, Jeannie type-checks the C code as if the Java expression had the equivalent C type, see Section 3.2.2 [Type equivalences], page 19. Jeannie also checks that the C code treats the value of the Java expression as an r-value, and in particular, does not assign to it. When a Java `return` statement returns from a C function, Jeannie type-checks the return value against the return type of the function as if it had the equivalent C type.

In order to contain nested Java code, the enclosing C code must be either part of a Java method, or must be in a C function that declares an explicit formal parameter `JNIEnv* env`. The `env` variable can also be used to facilitate incremental conversion of JNI code to Jeannie; other uses of this feature are discouraged.

If nested Java code contains any `break` or `continue` statements, those must not cross the language boundary, neither must they cross the boundary of a `with` statement.

3.2.5 with

NAME `_with` – Access entire Java array or string from C.

SYNTAX RULES

C.Statement	+=	. . . / `_with (` *WithInitializer* `)` *C.Block*
WithInitializer	=	*C.AssignmentExpression* / *C.Declaration*

SYNTAX NOTES

You can also write the keyword (`_with`) without a leading underscore (`with`). The leading underscore is mandatory only if you run the Jeannie compiler with the `-underscores` command line option.

An example for a `with` statement is `with (‘int* ca = ‘ja) { sendMsg(ca); }`.

An example for a C assignment expression is `msg->data = ‘ja`.

An example for a C declaration is `‘int* ca = ‘v.toArray()`.

DYNAMIC SEMANTICS

Jeannie's `with` statement accesses a Java string or array from C code like a C array in a well-defined scope. For example,

```
_with (‘int* ca = ‘ja) {
  for (‘int i=0, n=‘ja.length; i<n; i++)
    s += ca[i];
}
```

acquires a copy of Java array `ja`'s contents, sums up its elements, and then releases the copy while also copying back the contents. In the example, array `ja` is released when control reaches the end of the block. In general, the Java string or array is released when control leaves the body of the `with` statement for any reason, including return statements and exceptions. In the case of an exception, all modifications to the array are canceled, in other words, the original Java array is unmodified. When there is no exception, any changes to the C array are copied back into the Java array.

If the Java string or array is null, the `with` statement signals a `NullPointerException`. Otherwise, it initializes the C array to point to a copy of the Java array. Strings are UTF-8 encoded if the C array is of type `‘byte*`, and UTF-16 encoded if the C array is of type `‘char*`. Independent of encoding, modifying a string leads to undefined behavior.

Static semantics

The *WithInitializer* must be either a simple assignment to a C pointer variable, or a declaration of a C pointer variable. For the purpose of this discussion, let *ca* be the name of the C pointer variable, and let *ja* refer to the Java string or array on the right hand side of the *WithInitializer*. In the assignment case, *ca* must be modifiable, i.e., not `const`.

Let *CT* be the type of *ca*, and *JT* the type of *ja*. If *ja* is an array, then *JT* is `j`*E*`Array` for some element type *E*, and *CT* must be `` ` ``*E*`*`, for the same *E*. For example, if *JT* is `jintArray`, then *E* is `int`, and *CT* must be `` `int* ``. If `ja` is a string, then *JT* is `` `String ``, and *CT* must be either `` `byte* `` or `` `char* ``.

3.2.6 cancel and commit

Name `_cancel` / `_commit` – Release a Java array and discard / preserve changes.

Syntax rules

C.Statement	+=	. . .
	/	`_cancel` *C.Identifier* `;`
	/	`_commit` *C.Identifier* `;`

Syntax notes

You can also write the keyword (`_cancel` or `_commit`) without a leading underscore (`cancel` or `commit`). The leading underscore is mandatory only if you run the Jeannie compiler with the `-underscores` command line option.

An example for a `cancel` statement is `_cancel ca;`.

An example for a `commit` statement is `_commit ca;`.

Dynamic semantics

The `commit` and `cancel` statements initiate an abrupt control transfer to the code immediately following the `with` statement that initializes the named C pointer variable. A `commit` statement copies any changes back into the Java array, whereas `cancel` discards them. Both `commit` and `abort` release any resources necessary for implementing the `with` statement, notably the copy's memory.

Static semantics

The identifier must be the formal of a directly enclosing `with` statement.

3.3 Builtin functions

This section describes special C functions built into Jeannie. Builtin functions are recognized by the Jeannie compiler, which analyzes them and translates them specially. For example, the compiler enforces special constraints when it analyzes a builtin, such as matching a C buffer type to a Java array type.

You can write all of these builtins either with or without a leading underscore (e.g., `copyFromJava` vs. `_copyFromJava`). The leading underscore is mandatory only if you run the Jeannie compiler with the `-underscores` command line option.

3.3.1 copyFromJava

NAME `_copyFromJava` – Copy string or array elements from Java to C.

SIGNATURE

`‘int _copyFromJava(`*CT* `ca, ‘int ci,` *JT* `ja, ‘int ji, ‘int len)`

DESCRIPTION

Copy from `ja[ji...ji+len-1]` to `ca[ci...]`, and return the number of elements copied into `ca`.

If `ja` is an array, then *JT* is `j`*E*`Array` for some element type *E*, and *CT* must be `‘`*E*`*`, for the same *E*. For example, if *JT* is `jintArray`, then *E* is `int`, and *CT* must be `‘int*`.

If `ja` is a string, then *JT* is `‘String`, and *CT* must be either `‘byte*` or `‘char*`. If *CT* is `‘byte*`, then the copy involves a conversion from UTF-16 to UTF-8. This conversion may cause the return value (number of elements copied into `ca`) to differ from the `len` parameter (number of elements copied out of `ja`).

PARAMETERS

CT `ca`	C array that receives the copy.
`‘int ci`	Index in the C array where the copy starts.
JT `ja`	Java string or array from which the copy originates.
`‘int ji`	Index in the Java string or array where the copy starts.
`‘int len`	Number of copied elements from the Java string or array.

RETURNS `‘int` – Number of elements copied into the C array `ca`.

EXCEPTIONS

If one of the indices in the Java string or array `ja` is invalid, `copyFromJava` raises a `StringIndexOutOfBoundsException` or `ArrayIndexOutOfBoundsException`. If one of the indices in the C array `ca` is invalid, `copyFromJava` exhibits undefined behavior. To avoid a buffer overrun related to unicode conversion (from a Java string to a C `‘byte*`), you should call `stringUTFLength` before calling `copyFromJava`.

3.3.2 copyToJava

NAME `_copyToJava` – Copy array elements from C to Java.

SIGNATURE

`‘int _copyToJava(`*JT* `ja, ‘int ji,` *CT* `ca, ‘int ci, ‘int len)`

DESCRIPTION

Copy from `ca[ci...ci+len-1]` to `ja[ji...]`, and return the number of elements copied. The type *JT* must be `j`*E*`Array` for some element type *E*, and *CT* must be `‘`*E*`*`, for the same *E*. For example, if *JT* is `jintArray`, then *E* is `int`, and *CT* must be `‘int*`. The type *JT* must not be `‘String`, because strings are immutable in Java, and therefore, it does not make sense to copy elements into them.

Parameters

JT `ja`	Java array that receives the copy.
‘`int ji`	Index in the Java array where the copy starts.
CT `ca`	C array from which the copy originates.
‘`int ci`	Index in the C array where the copy starts.
‘`int len`	Number of copied elements from the Java string or array.

Returns ‘`int` – Number of elements copied into the Java array `ja`.

Exceptions

If one of the indices in the Java array `ja` is invalid, `copyToJava` raises an `ArrayIndexOutOfBoundsException`. If one of the indices in the C array `ca` is invalid, `copyToJava` exhibits undefined behavior.

3.3.3 newJavaString

Name `_newJavaString` – Create a new Java string from C.

Signature

‘`String _newJavaString(const` *CT* `ca)`

Description

Create a new Java string with the same contents as the C array `ca`. The type *CT* of the C array must be either ‘`byte*` or ‘`char*`. If *CT* is ‘`byte*`, then the string creation involves a conversion from UTF-8 to UTF-16. In either case (‘`byte*` or ‘`char*`), the C array must be null-terminated.

Parameters

CT `ca`	C array from which to copy characters into the newly allocated Java string.

Returns ‘`String` – Newly created Java string with the same contents as `ca`.

Exceptions

If the Java virtual machine does not have enough memory available to allocate the Java string, then `newJavaString` raises an `OutOfMemoryError`.

3.3.4 stringUTFLength

Name `_stringUTFLength` – Count length of Java string in UTF-8 representation.

Signature

‘`int _stringUTFLength(‘String js [ , ‘int ji, ‘int len ] )`

Description

Count how long the UTF-8 representation of the UTF-16 string `js` is. If the optional parameters `ji` and `len` are specified, count how long the UTF-8 representation of the substring `js[ji...ji+len-1]` is. You should use this function to find out how large a C ‘`byte*` buffer you need when copying (parts of) Java strings to C.

Parameters

`‘String js`
: Java string to measure.

`‘int ji` Start index of the region to measure.

`‘int len` Length in UTF-16 characters of the region to measure.

Returns `‘int` – Length in UTF-8 characters.

Exceptions
None.

3.4 Tools

This section describes the command line tools for compiling Jeannie programs. In the normal case, you should only need to use one of them: the “master script” `jeannie.sh`. It orchestrates the other Jeannie tools (preprocessor, compiler, postprocessor) as well as external tools (C and Java compilers).

3.4.1 jeannie.sh

Name `jeannie.sh` – Jeannie compiler master script.

Synopsis `jeannie.sh` [*options*] *file* [*c-files...*]

Parameters

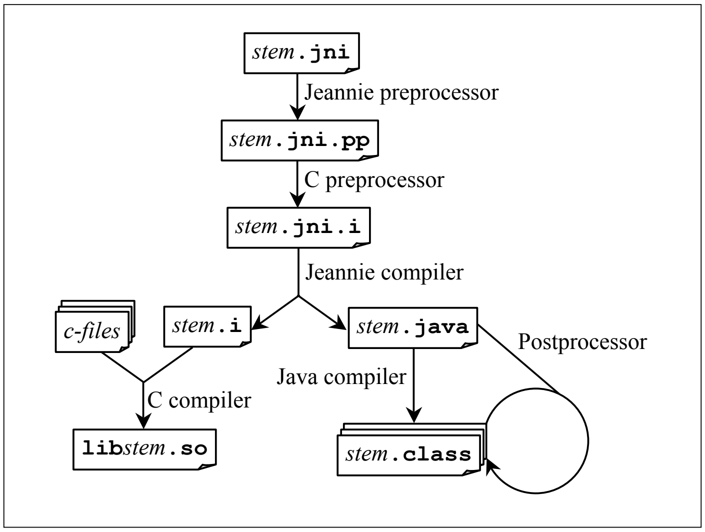
options Options may be in any order. See Options below.

file Main source file to compile, usually with the extension `.jni`. The *file* name should include the package directory. For example, if you compile a class `a.b.C`, where `a.b` is the package name, the file name should be `a/b/C.`*ext*. See also the `-sourcepath` option below.

c-files... Other files to compile and link with the C compiler (may be C sources or objects). These are added into the dynamically linked library created from the C part of the main source file.

Description
The `jeannie.sh` master script compiles a Jeannie source file into Java class files and a dynamically linked library. It does this by calling other tools that

transform the file through a number of stages. The following picture illustrates this:



The `jeannie.sh` script first splits *file* (the name of the main source file) into a *stem* and an extension. The extension specifies the start *stage* of the compilation. For example, if the command line is

```
jeannie.sh Main.jni.pp
```

then *stem* is `Main` and the extension is `jni.pp`. Hence, processing starts at stage `jni.pp`, and the first processing step runs the C preprocessor. By default, `jeannie.sh` follows all processing steps from the start stage to the end. The `-stopAfter` option overrides this default by specifying a stop stage. For example, if the command line is

```
jeannie.sh -stopAfter i,class Main.jni
```

then processing stops after `Main.i` and `Main.class` have been generated. In other words, `jeannie.sh` does not run the C compiler to create a dynamically linked library.

Here is a brief description of each processing step:

Jeannie preprocessor
: Inject Jeannie-specific definitions at the start of the file.

C preprocessor
: Resolve `#include` and other directives and expand macros.

Jeannie compiler
: Translate Jeannie code into separate C and Java code.

C compiler
: Typically `gcc`, compile C code to dynamically linked library. The file name of the generated library depends on the platform:

stem`.dll` on Cygwin, `lib`*stem*`.so` on Linux, and `lib`*stem*`.jnilib` on Mac OS.

Java compiler
: Typically `javac`, compile Java code to class files.

Postprocessor
: Inject `//#line` directives from `.java` file into `.class` files.

OPTIONS

`-cc` *path*
: File name of your C compiler. Overrides the `CC` environment variable. If not specified, `jeannie.sh` uses the `gcc` executable it finds in your `PATH`.

`-cp | -classpath` *paths*
: Search existing user class files in *paths*. Overrides the `CLASSPATH` environment variable. This is a list of directories or jar files, separated by colons (on Linux or Mac OS) or semicolons (on Windows). It must include the xtc root directory in order to find the classes that implement the Jeannie compiler.

`-d | -destpath` *dir*
: Write output generated files to *dir*. Specifically, if you compile a class `a.b.C`, where `a.b` is the package name, the generated files will have names based on *dir*`/a/b/C.`*ext*.

`-flattenSmap`
: Rewrite line numbers with SMAP, do not stratify. The Jeannie compiler changes the symbol information in generated class files to refer to the original `.jni` source file to enable source-level debugging. The `-flattenSmap` option determines whether this is accomplished by erasing the line number information or by adding an additional source map stratum as specified by JSR-45 `http://jcp.org/en/jsr/detail?id=45`. The difference becomes visible for tools that do not yet support JSR-45, such as Java virtual machines printing exception backtraces using the line numbers of the Java source file.

`-g`
: Produce debugging symbols. Can not be used in conjunction with the `-pretty` option. The `-g` option is passed through to the C compiler as well as the Java compiler.

`-h | -help`
: Print a short summary of the command line options of `jeannie.sh`.

`-I`*dir*
: Search header files in *dir*. This option is passed through to the C preprocessor, which uses it to resolve `#include` directives.

`-in` *dir*
: See the `-sourcepath` option below.

`-javaHome` *dir*
: Use the JDK installed in *dir*. Overrides the `JAVA_HOME` environment variable. If not specified, `jeannie.sh` infers this directory based

on where it finds the `java` executable in your `PATH`. The Jeannie compiler looks for `javac` and `java` in *dir*`/bin`.

`-jniCall` *qualifier*
: Prepend generated C JNI functions with *qualifier*. Overrides the `JNI_CALL` environment variable. This is the expansion of the `JNICALL` macro defined in `jni.h`, which specifies the calling conventions on platforms where that matters. You should not need to specify this option, as `jeannie.sh` will infer it for you. It is typically the empty string on Linux or Mac OS, and the string "`__attribute__((__stdcall__))`" on Cygwin.

`-l`*library*
: Search *library* when linking with the C compiler. This option is passed through to the C linker, which uses it to resolve external symbols. For example, `-lm` specifies the math library (`m`) to search for mathematical functions such as `sqrt` and `cos`.

`-nowarn`
: Disable compiler warning messages. By default, `jeannie.sh` invokes the C compiler with `-Wall`; the `-nowarn` option overrides this default. Also, `-nowarn` gets passed through to the Java compiler.

`-platform` *platform*
: Compile for *platform*. Must be one of `Cygwin`, `Linux`, or `MacOS`. Usually, `jeannie.sh` will infer the platform for you, but if it can't, you need to specify it on the command line. Note that this option is not sufficient for cross-compiling, as the compilation also depends on the installed C compiler, header files, and libraries.

`-pretty`
: Optimize generated code for human readability. Can not be used in conjunction with the `-g` option. By default, `jeannie.sh` intersperses generated Java code with line markers such as

```
//#line 7 Main.jni
```

Line markers are then uses to support debugging at the level of the original source code, in this case, `Main.jni`. The `-pretty` option suppresses line markers and leads to more natural indentation. That is useful when you need to inspect generated source code by hand.

`-underscores`
: Require leading underscore in keywords, e.g., `_with`. By default, the Jeannie preprocessor defines aliases for keywords and builtin functions that omit the leading underscore. But this can lead to name clashes with included header files. When that happens, you can resolve the name clash by specifying the `-underscores` option and writing all Jeannie keywords and builtins with underscores.

`-sourcepath` *dir*
: Read input source files from *dir*. Specifically, when you compile a class `a.b.C`, where `a.b` is the package name, the source file should reside in *dir*`/a/b/C.jni`.

`-stopAfter` *stage*
: Stop compiling after reaching *stage*. For example, if *stage* is `jni.i`, then `jeannie.sh` will run the Jeannie preprocessor and the C preprocessor, and then stop. See the DESCRIPTION above for the full list of stages. By default, if `-stopAfter` is not specified, `jeannie.sh` will run all stages.

`-v | -verbose`
: Print commands executed by `jeannie.sh`. Each command is prepended by the source location in `jeannie.sh` just before running it. Also, each command may print its own messages, such as a copyright notice.

`-verboseSettings`
: Print internal settings of this bash script. This option is useful when options or environment variables (see ENVIRONMENT below) do not have the desired effect.

`--`
: Treat remainder of command line as file names. This option is useful when one of your file names starts with a dash (`-`), and might therefore be mistaken with a command line option otherwise.

ENVIRONMENT

`CC`
: Name of your C compiler executable. See the `-cc` command line option above for details.

`CLASSPATH`
: Paths where to search existing user class files. See the `-classpath` command line option above for details.

`JAVA_HOME`
: Path where the JDK is installed. See the `-javaHome` command line option above for details.

`JNI_CALL`
: Qualifier to prepend in front of C JNI functions. See the `-jniCall` command line option above for details.

3.4.2 Preprocessor

NAME `xtc.lang.jeannie.Preprocessor` – Inject Jeannie-specific definitions.

SYNOPSIS `java xtc.lang.jeannie.Preprocessor` [*options*] *file*

PARAMETERS

options
: Options may be in any order. See OPTIONS below.

file
: Main source file to compile, usually with the extension `.jni`. If you omit the file name, the preprocessor prints a description of the command line options.

DESCRIPTION

The Jeannie preprocessor injects Jeannie-specific definitions at the start of the input *file*, and writes the result to `stdout`. This usually gets invoked from the `jeannie.sh` master script, but you can also run it stand-alone. In the usual case, the input file would have extension `.jni`, and you would pipe the output to a file with the extension `.jni.pp`. The injected definitions appear at the start of the initial '`.C{...}` block, which means they precede any other C declarations that you put there either directly or with `#include`.

Options

`-silent` Enable silent operation. This suppresses the boilerplate tool name and copyright notice that the preprocessor emits otherwise every time you run it.

`-underscores`
Require leading underscore in keywords, e.g., `_with`. By default, the Jeannie preprocessor defines aliases for keywords and builtin functions that omit the leading underscore. But this can lead to name clashes with included header files. When that happens, you can resolve the name clash by specifying the `-underscores` option and writing all Jeannie keywords and builtins with underscores.

Environment

`CLASSPATH`
Paths where to search existing user class files. Must include the xtc root directory to find the preprocessor itself.

3.4.3 Compiler

Name `xtc.lang.jeannie.Jeannie` – Translate Jeannie to Java and C source.

Synopsis `java xtc.lang.jeannie.Jeannie` [*options*] *file*

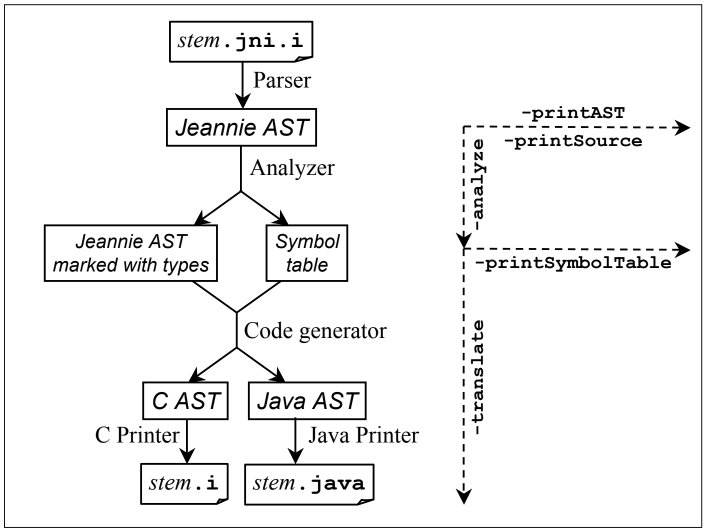
Parameters

options Options may be in any order. See Options below. Usually, you would specify `-analyze -translate` to run both the semantic analyzer and the code generator.

file Main source file to compile, usually with the extension `.jni.i`. In general, you must run the Jeannie preprocessor and the C preprocessor first, otherwise, many of the C identifiers in the file are undeclared and lead to errors from the C semantic analyzer.

Description
The Jeannie compiler translates Jeannie source code into separate Java and C source code. This usually gets invoked from the `jeannie.sh` master script, but you can also run it stand-alone. In the usual case, the input would have extension `.jni.i`, and the compiler generates two files with the same stem and extensions `.i` (for preprocessed C code) and `.java` (for Java code). The options

serve to run the compiler only partially and to print intermediate results. The following picture illustrates how the compiler works internally.



A detailed technical description of the Jeannie compiler internals is in the conference paper at http://cs.nyu.edu/rgrimm/papers/oopsla07.pdf.

OPTIONS

`-analyze` Analyze the program's AST. Required for `-translate` or `-printSymbolTable`. Runs the Jeannie semantic analyzer on the abstract syntax tree, which includes both C and Java type analysis.

`-in` *dir* Add the specified directory to the file search path. When the compiler encounters a reference to a class that is not defined in the current file, it searches for other files that define it. This search starts in source files of the form *dir*/*package*/*class*`.java`. If it fails to find the class this way, the compiler attempts to find a compiled version of the class based on reflection and the `CLASSPATH` environment variable.

`-jniCall` *word*
Prepend generated C JNI functions with *qualifier*. This is the expansion of the `JNICALL` macro defined in `jni.h`, which specifies the calling conventions on platforms where that matters. Defaults to the empty string, which is correct on Linux or Mac OS; on Cygwin, you should provide set it to "`__attribute__((__stdcall__))`".

`-out` *dir* Use the specified directory for output.

`-pedantic`
Enforce strict C99 compliance.

`-pretty` Optimize output for human-readability. By default, the compiler intersperses generated Java code with line markers such as

```
//#line 7 Main.jni
```

Line markers are then uses to support debugging at the level of the original source code, in this case, `Main.jni`. The `-pretty` option suppresses line markers and leads to more natural indentation. That is useful when you need to inspect generated source code by hand.

`-printAST`
Print the AST in generic form. This will usually create a lot of output on `stdout`. Looking at the abstract syntax tree helps compiler hackers validate their assumptions when crafting visitors.

`-printSource`
Print the AST in Jeannie source form. The result should be more or less the same as the input, with different indentation and without comments.

`-printSymbolTable`
Print the program's symbol table. Requires option `-analyze`.

`-silent` Enable silent operation. This suppresses the boilerplate tool name and copyright notice that the compiler emits otherwise every time you run it.

`-strict` Enforce strict C99 compliance.

`-translate`
Generate separate C and Java code. Requires option `-analyze`.

ENVIRONMENT

`CLASSPATH`
Paths where to search existing user class files. Must include the xtc root directory to find the compiler itself.

3.4.4 Postprocessor

NAME `xtc.lang.ClassfileSourceRemapper` – Add debugging symbols to classes.

SYNOPSIS `java xtc.lang.ClassfileSourceRemapper` [*options*] *source-file class-file*

PARAMETERS

options Options may be in any order. See OPTIONS below.

source-file Java source code file from which to extract line number information. This is typically automatically generated by the Jeannie compiler.

class-file Java bytecode file to which to add line number information. This is typically generated by the Java compiler translating the *source-file*.

DESCRIPTION

The Jeannie postprocessor augments class files with symbolic information. This usually gets invoked from the `jeannie.sh` master script, but you can also run it stand-alone. The postprocessor reads the input *source-file*, which contains line markers such as

```
//#line 7 Main.jni
```

Line markers map lines in the generated Java source file back to the original Jeannie source file. The postprocessor injects this information into the *class-file*. That is useful for source-level debugging and for exception backtraces.

OPTIONS

`-flatten` Append an SMAP to the end of the class file as "SourceDebugExtension". This is the default. The SMAP (source map) format (specified in JSR-45 `http://jcp.org/en/jsr/detail?id=45`) is a general and powerful way to provide remapping information for source-to-source transformations. It works well with the current Java debuggers (SUN jdb 1.6 and the eclipse 3.2 Java debugger). However, JVMs in the SUN JDK 1.6 and IBM J9 1.5.0 do not use the SMAP when dumping stack traces for exceptions.

`-stratify`
: Rewrite the "LineNumberTable" attribute for each method in the class file, and modify the "SourceFile" attribute. This has an advantage of working well with both the current Java VMs and debuggers. However, this does not work if the number of the original source files is more than one.

ENVIRONMENT

`CLASSPATH`
: Paths where to search existing user class files. Must include the xtc root directory to find the postprocessor itself.

Index